WestminsterResearch
http://www.wmin.ac.uk/westminsterresearch

Improving quality of service in application clusters.

Sophia Corsava
Vladimir Getov
Harrow School of Computer Science

Copyright © [2003] IEEE. Reprinted from International Parallel and Distributed Processing Symposium (IPDPS'03): proceedings, pp.253-260.

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University of Westminster's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this document, you agree to all provisions of the copyright laws protecting it.

The WestminsterResearch online digital archive at the University of Westminster aims to make the research output of the University available to a wider audience. Copyright and Moral Rights remain with the authors and/or copyright owners. Users are permitted to download and/or print one copy for non-commercial private study or research. Further distribution and any use of material from within this archive for profit-making enterprises or for commercial gain is strictly forbidden.

Whilst further distribution of specific materials from within this archive is forbidden, you may freely distribute the URL of WestminsterResearch. (http://www.wmin.ac.uk/westminsterresearch).

In case of abuse or copyright appearing without permission e-mail wattsn@wmin.ac.uk.

Improving Quality of Service in Application Clusters

Sophia Corsava and Vladimir Getov
Harrow School of Computer Science, University of Westminster, London, U.K.
Email: sophiac6@yahoo.com, V.S.Getov@westminster.ac.uk

Abstract. *Quality of service (QoS) requirements, which include availability, integrity, performance and responsiveness are increasingly needed by science and engineering applications. Rising computational demands and data mining present a new challenge in the IT world. As our needs for more processing, research and analysis increase, performance and reliability degrade exponentially. In this paper we present a software system that manages quality of service for Unix based distributed application clusters. Our approach is synthetic and involves intelligent agents that make use of static and dynamic ontologies to monitor, diagnose and correct faults at run time, over a private network. Finally, we provide experimental results from our pilot implementation in a production environment.*

Keywords: Application clusters, distributed applications, quality of service, performance.

1. Introduction

Researchers, analysts, scientists and engineers, need reliable and powerful systems. Having the ability to run multiple analyses, experiments and realistic simulations can lead to new and more comprehensive discoveries. Research scientists and engineers can properly research by performing experiments in a trial and error mode. Most researchers are led to the wrong conclusions as they are faced with processing problems. To overcome them, they reduce sample populations. Alternative scenarios cannot be investigated thoroughly and creativity cannot be fully expressed. Delivery of processing outputs and timing get adversely affected as well. Opportunities get lost this way too, if it takes months to determine which is the best processing technique for atomic energy for example. Data mining techniques cannot be put to their full use, as by nature they are processing intensive. The majority of database servers cannot withstand the load of running repeated comparisons of large data groups against a set of possible parameters and outcomes. Processing needs to be smooth, transparent and efficient.

Robust, dependable infrastructures are very difficult to maintain [1]. Complex environments are difficult and costly to manage and, generally speaking, complexity reduces the predictability and reliability of application services and systems [12, 17]. Due to this complexity, it is usually quite difficult to identify performance problems, bottlenecks or failures promptly. Analysing collected availability and performance data can be a time-consuming thankless task that requires pain-staking manual labour. Deciding upon a course of action to resolve failures and performance problems may take weeks and can be very costly. With the exponential growth of distributed clusters and emerging grids, the matching human component is simply not available [3, 4, 12]. We urgently need to reconsider the way we build infrastructures and troubleshoot faults. Infrastructures, (hardware, network and software) are essential prerequisites for efficient service delivery. In this paper, we propose a solution to these problems by introducing the architectural framework for a fault-tolerant, self-managing, intelligent infrastructure for Unix based distributed application clusters.

This paper is organized as follows. In section 2 we discuss related work. In section 3 our building methodology and approach, while section 4 discusses some preliminary results from one of our implementations at a UK-based financial customer site.

2. Related Work

Structured troubleshooting and fault correction approaches are widely used in the application domain. These techniques include recursive restarts [5], check pointing [18], reboot [12] and undoing old configurations [12]. The check-pointing technique allows applications to recover from the last point of failure by copying on a regular basis their status on stable storage and then retrieving it. Application recursive restarts are based on the principle of infrastructure-centric software design: move intelligence from endpoints into the supporting infrastructure. Reboot, restarts not only the application but also the underlying operating system and undoing old configurations involves restoring old backups and overwriting current assumed “invalid” settings. A newer approach is the N-layered architecture for application

development that allows for resiliency and better performance [2].

A lot of important work has been done in the areas of fault diagnosis, performance and decision-making. Current diagnostic methods include: the threshold analysis, the bottleneck analysis, the what's different analysis and the correlation analysis [9, 10]. Closely related to our project is also the very important work done by John Wilkes and R. Golding on self-managing, self-configuring storage [8, 19]. In addition, fault and/or decision trees are commonly used to diagnose faults and action corrective measures.

There are a number of tools that measure performance and monitor systems. These include BMC, HPGlance Plus, HP Measureware, SystemEdge, Sun Management Centre, TeamQuest, Landmark Performance Works, Aurora Software Sarcheck, Foglight Software RAPS, Compuware Ecotools, Datametrics Viewpoint, Metron Athene, Network Weather Service (mostly for networks) etc [6]. To our knowledge, there are no commercial tools that automatically correct performance problems.

3. Building Methodology

3.1 Overview

Our approach involves:

1. Unix shell based intelligent agents that monitor, troubleshoot and manage distributed services within the datacentre. These agents also collect detailed system performance/availability measurements.
2. Dedicated administration servers that act as external agent coordinators in a high-availability failover configuration and share a common pool of NFS mounted disks, to avoid single points of failure.
3. A dedicated private network, where all agent-related traffic goes through to avoid congesting the public LAN.
4. Static and dynamic ontologies [7, 15]. These ontologies include:
   a. Index static service lists (ISSL) that contain very basic information about each server or resource IP address and services. They can contain up to 200 entries and are manually updated.
   b. Dynamic local service profiles (DLSP) that are generated by each agent controlled server in regular intervals and contain information about server hardware, software, load, capacity and services.
   c. Static local knowledge templates (SLKT) that contain information about what the server should be like hardware-wise, which applications it should run, all application external and internal dependencies and requirements (file systems, path names, application component startup sequences, binary location, application type, version, name, IP address, port it listens to – if any, application process names and numbers, etc.).
   d. Dynamic global service profile lists (DGSPL) that contain information about all running and available services across the entire datacentre. Available services are presented by <Server type, OS, memory and CPUs, Application type and version, Current Load, Users logged in, Geographical Location, Site Name>. These lists can also be used to present services to grids.

3.2. Assumptions

We make the following assumptions:

- All servers are Unix-based and NTP (time) synchronized.
- For all applications participating in the datacentre (databases, web servers etc), we have startup and shutdown scripts.
- Specialized application developers have provided us with tools and methodologies to test and confirm if these applications are up and running and available to be used, that have been incorporated in the agent source code and ontology definition. In addition these people have provided us with application specific connectivity time-out definitions.
- All ontologies, templates, files and logs produced are flat ASCII files generated by I/O Unix pipes, readable by most Unix tools (operators) and human administrators.
- The maximum load a server can successfully sustain has been provided to us by either hardware/operating system manufacturers combined with practical experience information by human experts and adapted based on our own observations.
- All communications are based on TCP/IP.
- We use the term application/service interchangeably, as in the context of this paper, applications are service vehicles.
- We assume that the majority of distributed services run on physically separate Unix servers.

3.3 Intelligent agents

Intelligent agents [20] or intelliagents are Unix programs that monitor systems and services and wherever possible automatically correct run-time operational faults with as little downtime as possible. They are also responsible for collecting detailed performance, load and availability measurements for systems and services. They can be

IEEE COMPUTER SOCIETY

thought of as huge wrappers that can be used to administer, maintain and troubleshoot every single infrastructure aspect. Intelliagents use constraint-based causal reasoning [13]. The data structures they use are flat ASCII textual ontologies which contain minimum and maximum software and hardware related variables, as well as application information. Our static ontologies represent the constraints in the reasoning. Intelliagents are not memory resident. They are highly modular. They are installed locally on each server they monitor, always in the same physical location *"/apps/intelliagents"*. They are "awakened" every X minutes (every 5 minutes for example) by local to each host Unix crons. Intelliagents do not use a relational database (to avoid corruptions and for simplicity), they use static ontologies in the form of static knowledge templates and service lists to generate dynamic ones. Ontologies are being used in logic, mathematics and Artificial Intelligence. An ontology is "a description (like a formal specification of a program) of the concepts and relationships that can exist for an agent or a community of agents" [7]. The subject of *ontology* is "the study of the *categories* of things that exist or may exist in a domain" [15].

All intelliagent related communication goes through the private agent network to avoid putting any performance/load overheads to the public LANs (see Figure 1). All participating devices and resources in the datacentre are connected to the private agent network and one or more public LANs. If the private network fails, intelliagents can automatically re-route their communication traffic over the public LAN, using Unix administration commands.

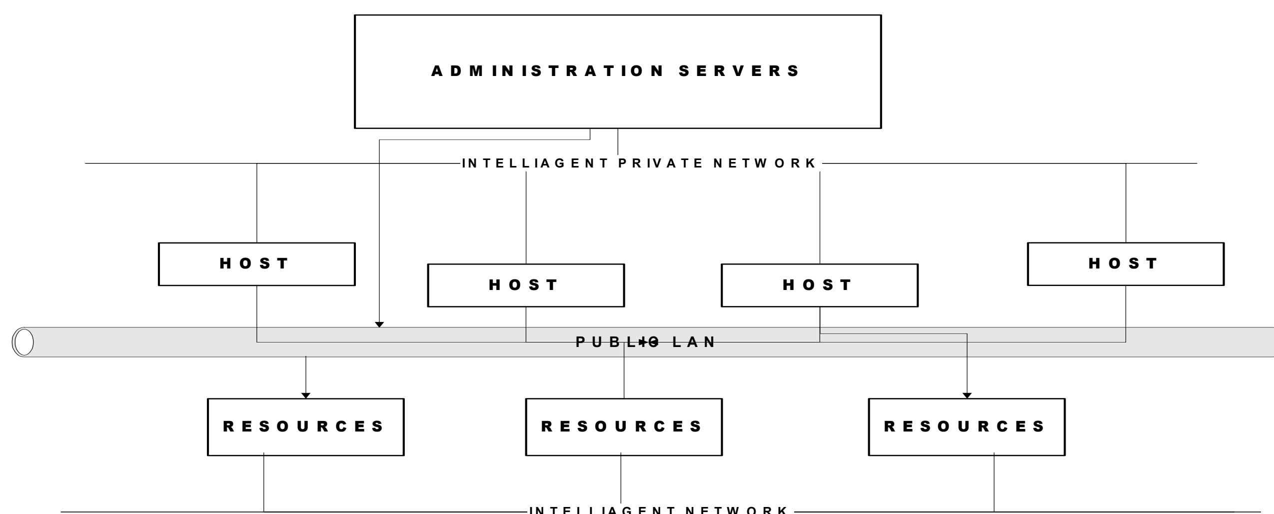

Figure 1. A hi-level view of the intelliagent private network and administration servers. All intelliagent communications go through the private intelliagent LAN to avoid loading public LANs.

Whenever a local intelliagent runs, it produces a flag in the dedicated *"/logs/intelliagents/intelliagent_name"* directory on the local server disk to show the status of the run. A number of flags are produced with appropriate naming conventions that show what happened and exactly where the agent found a fault. Absence of these flags means that we either have an internal intelliagent problem or that they did not run at all. Administration servers monitor the creation of these flags every X+5 minutes, where X is the frequency intelliagent run, i.e. every 10 minutes (adjustable parameter). If these flags are not there, they start troubleshooting intelliagent processes. For each component there is one special intelliagent (such as one for the CPU, one for the network card etc). Whenever an agent detects an error it tries to fix it. All intelliagents run in parallel, in a distributed manner and do not depend on each other. At startup each intelliagent checks to see if any other of the same type is running, if so it exits – i.e. one can never have two backup intelliagents running at the same time. It also removes flags from previous runs and old local dynamic service profiles. Intelliagents are monitored by dedicated external administration servers to ensure correct function.

For each application type there are customized error categories. Application health is determined by attempting to connect to them every Y minutes and run basic commands (such as a get on a web server process for example). This is essentially the way intelliagents communicate with applications – by trying to use them and read the resulting exit code in the Unix shell.

Each intelliagent has 5 major parts: a) Monitoring, b) Diagnosing, c) Self-Healing/Action/Repair, d) Communication/Logging, e) Self-maintenance. The monitoring part is tasked to look after one particular system resource or aspect. Whenever the monitored subject does not respond as expected, the diagnosing part

IEEE COMPUTER SOCIETY

is invoked and goes through a series of tests to determine the root of the problem. The diagnostic procedure is done in two ways; statically and dynamically. Statically, from parsing and examining error logs and dynamically by the use of Unix administration commands to ensure the best possible diagnosis. Based on these findings the self-healing portion gets activated and starts repairing the faults. The communication part is responsible for communicating with other intelliagents and human operators. It is also responsible for logging all intelliagent activities and results.

Self-maintenance is an integral part of all intelliagents and every time an intelliagent runs, it looks after its individual logs. Each of the five intelliagent parts can get activated or deactivated either during installation or subsequently.

Intelliagents are classified based on their functions and tasks. Intelligent categories include: 1) Hardware agents that look after hardware components (CPU, memory, boards etc), 2) Operating system/network agents that look after all OS and network related aspects, 3) Resource intelliagents that are responsible for managing and configuring resources such as disks, network cards, virtual memory etc, 4) Application/Service intelliagents that manage and troubleshoot local and global application/services across the datacentre, 5) Status intelliagents that dynamically generate status profiles for servers, resources and services in terms of availability, load, capacity and geographical location, and 6) Performance intelliagents that collect performance and availability logs. These intelliagents can suggest what may be wrong during service degradation and have limited troubleshooting capabilities.

3.4 Service management

Service management is handled in a what could be called unorthodox way. Each local server in the datacentre is responsible for "knowing" and taking care of its own resources and services. Its local status intelliagent is "awakened" by the Unix cron and compiles dynamically its local DLSP.

To confirm that local services are available on each server, the local status intelliagent invokes local service intelliagents who attempt to connect to local running services and perform very simple queries (e.g. in the case of a web server they do an http "get", for a database they connect and attempt to do a "select * from table name"). Connectivity tests and timeout baselines are provided by specialized application/service providers as discussed. If services that should be running on that server are not running, intelliagents start troubleshooting. Their aim is to ensure that local services run at all times and if not restart them. Once they achieve this, they perform the prescribed connectivity tests again and if there is a problem they cannot resolve they notify human administrators (usually via email or SMS).

Manually created ISSPs have been experimentally proven to be the best way to maintain server information, as datacentres do not undergo drastic reconfigurations in terms of existing devices. We have moved the responsibility of monitoring services to each server locally. This has been proven to be the safest and less "resource" expensive way to keep detailed information about servers, services and resources. In addition centralised management methodologies have been proven unsuccessful in big complex environments [11, 17].

3.5 Performance intelliagents

Our performance measurement techniques were orientated towards workgroup aggregation. We divided our measurements into 5 main groups: 1) Operating system, 2) Network, 3) Disks, 4) Application processes and 5) User processes. Measurements were kept in a special logs directory and were classified first by server name and then by measurement group. All measurements were recorded in ASCII text files, created by Unix pipes through standard output redirection. We observed 1) I/O rates on disks and network devices, 2) processes per user name, 3) per command name and arguments, 4) per user and command name, 5) per CPU and 6) the match between network packets, port numbers and protocols.

All techniques were non-intrusive as they did not load the system they were monitoring. For each monitored resource type or workgroup, a dedicated performance intelliagent was responsible for collecting performance statistics and comparing them against pre-scripted baseline thresholds, every 10 or 15 minutes. All collected data were manipulated as text strings. Different types of measurements were associated together by matching their timestamps. Measurements were ordered by timestamp and treated as a time series to produce graphical representations of the system performance either as a whole or by component/workgroup. Each file produced by persistent state processes, was managed as a circular queue, the length of which was configurable. Every time these intelliagents run, they produced flags to indicate what happened. Every time, a threshold was exceeded they notified us via email or SMS. The tools used, were standard Unix tools such as *vmstat, iostat, sar, netstat, nfsstat, top* etc [14]. To determine accurately the behaviour of each process, we used microstate measurements where applicable, as most modern CPUs allow for them. The accuracy of microstate measurements is microsecond resolution and the overhead is sub-microsecond (units are nanoseconds). In this way we had very accurate thread and process accounting. More about microstate accounting can be found in [6].

3.6 Baselines and thresholds

Baselines were set based on the hardware configuration of each system and the application type it was running. These baselines were determined with the help of hardware, operating system and application experts, who had given us expected application performance times as well as our own observations. Every time a baseline setting was not proven to be correct, we adjusted it accordingly. This happened quite often in the case of newly installed applications primarily. Utilisation spikes happened quite often, and in these cases we used our performance measurements to determine what was wrong. We had developed customised system builds for each hardware, operating system and application type and we made sure that all the servers in the datacentre were built as such. This policy was proven very effective as we avoided known pitfalls.

The measurements we considered for the operating systems were: 1) Memory “sr” (scan rate), “po” (page out), page faults and free memory measurements to determine memory shortage, 2) CPU run queue, to detect any processes waiting to be served by any CPU, 3) Overall CPU idle time %, 4) Blocked processes waiting for I/O, 5) Per process CPU and memory utilization, and 6) Disk I/O and throughput. We used 30 second intervals during I/O measurements to avoid spikes in the load. We were interested in the asvc_t and wsvc_t values (read and write response times).

For the network we considered: 1) Network interface utilisation statistics and errors, 2) Network route utilisation, 3) NFS statistics, 4) TCP/IP bandwidth and end-to-end round trip latency measurements, 5) Size of incoming/outgoing network packets and TCP windows, 6) Network connection time to live and 7) Name server response (DNS, NIS, NIS+, LDAP).

For databases, we used scripts that had a lot of input from experienced database administrators. We used a combination of Unix tools and SQL commands to monitor and measure their performance. We considered the following measurements: 1) Time taken for a request to connect to the database, 2) Time taken for the request to be served by the database, 3) Time taken for the database to initialise, 4) Time taken for the database to shutdown, 5) Time taken for the database backup to complete, 6) Per process CPU and memory utilisation, 7) Number of users connected to the database and for how long each, 8) Memory allocated at startup, 9) Database checkpoints and 10) Memory per transaction.

For web servers and application GUIs we considered the following measurements: 1) Time taken to connect to them, 2) Time taken for the process to come back with the results of the query, 3) Per process CPU and memory utilization, and 4) Number of http/application connections and for how long each.

For distributed applications we observed the time taken for a request to be served by the entire application from beginning to end. Every 15 to 30 minutes we initiated a dummy process to run through all application components, simulating a user and measure the total response time, in addition to the business-as-usual requests.

4. Results

One of the sites our work was implemented was a financial company of a UK based international customer. Servers included SUN, HP, IBM and linux machines. The breakdown of machines and their functions were: 100 database servers, a mixture of Oracle and Sybase databases, running on Sun Enterprise Series 4500, and E10Ks. 55 transaction processing servers a mixture of E10Ks, Ultra 10s, linux, E450s, E220Rs HP K and T series and 60 front-end application IBM SP2 servers for user front-end financial applications. Services were distributed across these servers. All data was residing on local disks. The network was 100 Base/T ethernet for all servers.

Financial analysts used services for data-mining, financial projections, financial model evaluations, market data/trend simulations and analytical reports. The Load Sharing Facility, LSF [16] application was used for scheduling jobs against databases. Users via the application GUI, manually selected database servers to submit jobs or submitted them to be processed at a latter time, using either native LSF utilities, or Unix utilities like “cron” or “at” jobs. Computations were deployed across different geographical sites the customer had. Market data feeds would come in from all parts of the world from international customer sites and other places such as Reuters.

The main problem the users had was that application components were failing very often and on many occasions errors were latent. They had no means to automatically correct operational faults. For them downtime meant severe financial losses as their systems needed to be available on a 24x7 basis. Stock brokers needed frequent access to databases, market trend analysis reports, forecasts, projection and simulation results, to decide how to handle end-customer investments. Because it was a high-pressure complex environment, downtime had big impacts on service integrity, safety and QoS. What was happening on a regular basis was that various application components would stop working altogether and operators did not know where to start looking. Large database jobs scheduled to run overnight would frequently crash databases and calculations would not complete. Human operators tried to resolve operational problems and faults manually. Available services would often become unavailable without any explanation and

users would become increasingly frustrated. The financial company would suffer financial losses because analysts and researchers could not easily quantify and qualify financial models and analyze market trends. Operators were under immense pressure to resolve operational faults and performance problems under difficult circumstances and usually during the night and the end of financial periods.

The users used for monitoring BMC patrol and SystemEdge [6]. The environment was highly complex and faults (operational, performance and human errors) would make things much worse. Experienced administrators were "on-call" every night. Operators had the main task of notifying system administrators about fatal faults. There were many time-delays caused by operators not understanding how critical a fault was, or trying to locate the on-call people during the night. It could take up to 2 hours at a time for a service or server restart, as faults had to be diagnosed and that was difficult as services were distributed. In addition, a number of people had to be notified about the problem before any decisive action was taken (i.e. a server reboot). Often experts from more than one areas had to be called in together to decide what caused the problem. If experienced support people could not diagnose and correct remotely a fault, they were obliged to come in to work. The whole troubleshooting procedure (and subsequent downtime) could take an average of 4 hours in such cases.

The end users were becoming increasingly frustrated as they were having more downtime they could afford and the failure of one of more distributed application components would impact on service integrity. In Figure 2, one can see the detailed breakdown in hours based on the type of errors that caused downtime at the customer site for 1 year, before we implemented our software. Total downtime was 550 hours from service-related faults. 345 hours caused by databases crashing in the middle of a job, 60 hours caused by human errors, 30 hours caused by LSF errors, 40 hours from front-end user application downtime, 10 hours caused by firewall configuration/network errors, 50 hours from performance-related errors and 5 hours from services being completely unavailable (corruptions, bugs etc) and 10 hours from all types of hardware errors. After our work was implemented, downtime went down to 31 hours in total. The error distribution was, 8 hours from firewall/network related errors, 6 hours from various hardware related errors, 2 hours from human errors, 9 hours from performance errors, 1 hour from LSF errors, 3 hours from service front-end errors, 2 hours from complete service unavailability errors and 8 hours from databases crashing in the middle of a job. Intelliagents were used to monitor, troubleshoot and analyse distributed application services, servers, and performance. They were also used to automatically monitor and reschedule batch jobs if these failed. The administration servers generated dynamic global service profile lists per database type every 15 minutes on average. They managed the LSF job-scheduling tool and presented the best available database server for the batch job in a shortlist, with the best choice always first. The LSF software was configured to allow a finite number of scheduled jobs per database server and user access rights were defined at the Unix, the application/database and the LSF levels. Only specific users would submit specific types of jobs to a pre-determined group of database servers.

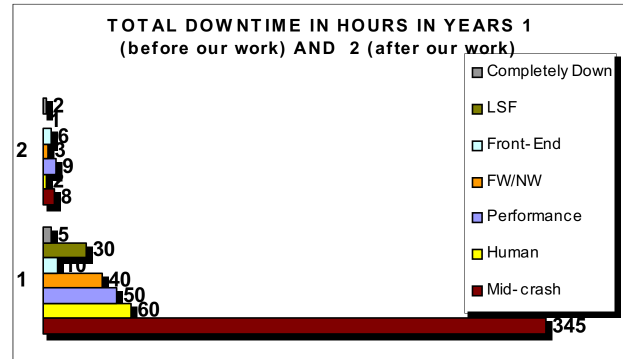

Figure 2. Breakdown of errors in hours and types for 2 years before and after our work was implemented. Provided by the customer.

Performance problems were detected and dealt with much faster, as performance intelliagents created comprehensive reports about what may have caused a performance related problem and helped narrow down various possibilities. The same applied for operational faults. Intelliagent error reporting mechanisms were integrated with SystemEdge and notifications were presented to operators from within the SystemEdge graphical user interface. Every time a fault was dealt with manually, we added a new troubleshooting procedure to the intelliagent source code and updated static ontologies accordingly when necessary.

Despite the impressive decrease in downtime our software was unable to take care of firewall/network and hardware related errors as well as eradicate completely human errors. Faults however, were detected within the first 5 minutes of them happening (the intelliagent run frequency), as opposed to about 1 hour during day time, about 25 hours over the weekends and 10 hours from overnight jobs (data provided by the customer using BMC Patrol). The customer did not have any means to automatically monitor, detect and correct any batch job failures or global distributed cluster faults prior to our work.

Intelliagents, in addition, checked every 5 minutes, if LSF processes were running (very often they would crash), if databases were up and running (likewise), the

IEEE COMPUTER SOCIETY

time batch jobs had left to complete (using pre-scripted LSF specific commands), if a batch job completed successfully (likewise), number of LSF scheduled jobs per database server as well as server, network and database load. They recorded all measurements and emailed summary reports to nominated administrators on a daily basis, on demand and whenever a job failed. Service intelliagents would attempt to troubleshoot and restart crashed processes and databases and they would notify human administrators accordingly. If jobs failed, intelliagents residing on the administration servers resubmitted them not based on the manual LSF settings and rules for job submissions, but based on the dynamically generated DGSPs. We had modified all ontologies to include LSF specific information such as number of jobs currently processed, jobs waiting to be processed and job number submission limit per database server. For this purpose we embedded LSF native commands in the service intelligent source code. The reason we chose to use DGSPs, was because every time a job crashed a database there were implicit conclusions that the user who submitted the job manually either a) did not select a powerful enough server, or b) selected a server that was already overloaded, or c) the server became overloaded later from scheduled job submission, or there were d) random or otherwise errors potentially responsible for database or server crashes.

These intelliagents, were also using SLKTs to select a server of equal or higher in power than the server that failed, i.e. if the failed server had 4 CPUs, 4 GBs of RAM and was a of a specific model, their selection process would "prefer" first a server of the same model with more CPUs and memory. Choosing "randomly" a server for resubmitting a failed job, without any knowledge of its past job submission history and failures, although not ideal, significantly decreased downtime from database crashes in the middle of a job. If intelliagents were unable to allocate a server for job submission at all for any reason, or if a server had crashed, they emailed human operators to manually troubleshoot the failed machine.

Figures 3 and 4 show respectively the average CPU and memory utilisation per system by intelliagents as opposed to BMC Patrol. Both cases clearly demonstrate the small percentage of system resources used on each system.

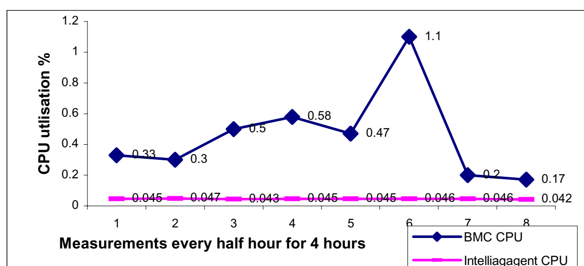

Figure 3. Intelliagent average CPU utilisation as opposed to BMC Patrol on a server at peak times.

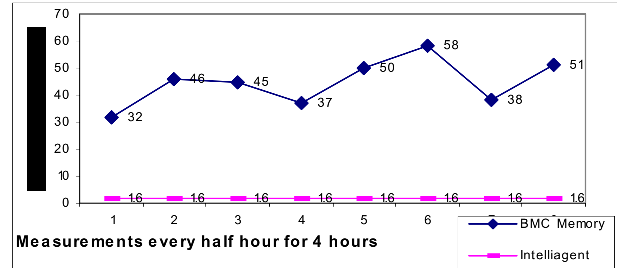

Figure 4. System Memory consumed by BMC patrol and intelliagents on the same server at peak time.

5. Conclusions and Future Work

Our approach has, experimentally, increased quality of service for Unix-based multi-component distributed services. From 550 hours of total downtime within a year before any of our parts of our work were implemented, downtime went down to 31 hours in total the next year.

Our main conclusions can be summarized as follows. In the case of complex multi-component applications local application-specific detection/correction mechanisms work much better than generic troubleshooting approaches. The distributed manner intelliagents work ensures that the more components an application is consisted of, the higher the probability is it will not fail if it is managed in this way. Agent code and ontologies are easily maintainable and do not tax the system they look after because of their size and simplicity. They are not memory resident and the flags they produce pinpoint errors accurately. Automated error detection and correction techniques improve quality of service as errors are picked up faster than ever before. Methodical, structured performance measurement and collection techniques can help resolve performance related issues and bottlenecks more efficiently and effectively. Administrators can generate timelines of system behaviour and observe similar behavioural patterns. In addition, they are notified automatically every time a threshold is exceeded.

Much work remains to be done, so our error detection and correction techniques are further improved and become more generic. We are also trying to reduce as much as possible manual input and generate automatically static ontologies. Performance modelling and dynamic troubleshooting of performance-related problems need further work. As we are not software developers, we used Unix shell languages as the best and easiest way to test our theories. The intelligent source code can be improved in that respect. Our approach, as is, at the moment cannot cater for network or obscure logical errors and needs manual input. It can however deal with latent errors up to a point, by restarting failed component applications. Extended logging of all system and

IEEE COMPUTER SOCIETY

intelliagent activities, ensure that human administrators have comprehensive information about all infrastructure aspects and can narrow down their search options when they do manual troubleshooting.

Our research continues in all these areas, in the hope that we'll improve our software and methodology further. Additional future work includes integrating our agent software with the grid technology. We hope the way agents generate dynamic global service lists (that contain information about all agent-enabled services) can be used in someway in the grid resource discovery and selection mechanisms for semantic grids. Intelliagents can manage quality of service very effectively for any distributed Unix application cluster, or standalone hosts by ensuring that all service components are available in the sequence they are meant to be. Service integrity and safety are protected in this way, as all interdependent distributed application components must be up and running for the distributed service to be considered healthy. If services are unavailable or take long to respond, intelliagents based on pre-scripted scenarios, try to restore them. If they are unable to do so, human administrators are notified so they can fix the problem. In that respect, services will not have resources vanishing unexpectedly without any explanations. Frequent monitoring ensures that service or server related faults are picked up and dealt with promptly.

References

1. Avizienis, J. -C. Laprie and B. Randell. "Fundamental concepts of dependability," Proc. of the 3rd Information Survivability Workshop, October 2000.
2. Chartier, Roger, "Application Architecture: An N-Tier Approach - Part 1", from http://www.15seconds.com/issue/011023.htm.
3. Corsava, Sophia, Getov, Vladimir, "Self-Healing Intelligent Infrastructure for computational clusters", SHAMAN workshop proceedings, ACM ISC conference, New York, June 2002.
4. Corsava Sophia, Getov Vladimir, "Intelligent Fault-Tolerant architecture for cluster computing", to appear at IASTED, PDCN03, Innsbruck, Austria, Feb 2003.
5. Candea George, Cutler James, Fox Armando, Doshi Rushabh, Garg, Priyank, Gowda Rakesh, "Reducing Recovery Time in a Small Recursively Restartable System", Proceedings of the International Conference on Dependable Systems and Networks (DSN-2002), Washington, D.C., June 2002.
6. Cockroft Andrew, "Sun Performance and Tuning", Talk, 2001.
7. Gruber, T.A. "A Translation Approach to Portable Ontology Specifications", 1993.
8. Golding Richard, Borowsky, Elizabeth, "Fault-tolerant replication management in large-scale distributed storage systems", Proceedings 18th IEEE Symposium on Reliable Distributed Systems, 1999.
9. Hellerstein, Joseph, "A comparison of Techniques for Diagnosing Performance Problems in Information Systems: Case Study and Analytic Models", IBM Research Division, 1994.
10. Hellerstein, J, Y. Diao, and S. Parekh, "A First-Principles Approach to Constructing Transfer Functions for Admission Control in Computing Systems", IBM T. J. Watson Research Center, To appear in the Conference on Decision and Control, 2002.
11. Li, Ming, Tao, Wenchao, Goldberg, Daniel, Hsu Israel, Tamir, Yuval, ``Design and Validation of Portable Communication Infrastructure for Fault-Tolerant Cluster Middleware," IEEE International Conference on Cluster Computing, Chicago, IL, (September 2002).
12. Patterson, D. "A new focus for a new century: availability and maintainability >> performance," Keynote speech at USENIX FAST, January 2002.
13. Pearl Judea, "Reasoning with cause and effect", IJCAI Award Lecture, 1999.
14. Quigley, Ellie, "Unix Shells by example", Prentice Hall, 1999.
15. Sowa, John F., "Knowledge Representation: Logical, Philosophical, and Computational Foundations", Brooks Cole Publishing Co, 2000.
16. Songnian Zhou. "LSF: load sharing in large-scale heterogeneous distributed systems". In Proceedings of the Workshop on Cluster Computing, December 1992
17. Veritas Cluster Server, release 1.3.0, Veritas Software Corporation, 2000.
18. Wong, Kenneth F. and Franklin, Mark, "Checkpointing in Distributed Computing Systems ", Journal of parallel and distributed computing, vol. 35, 67-75, 1996.
19. Wilkes, John and Keeton, Kimberly, "Automating data dependability", 10th ACM SIGOPS European Workshop, 2002.
20. Weiss G. "Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence", MIT Press, 1999.